PROGRAMMING II
JAVA LANGUAGE

COURSE 4 – COLLECTIONS I

PREVIOUS COURSE CONTENT

- **Inheritance**
  - Abstract classes
  - Interfaces
  - instanceof operator
- **Nested classes**
- **Enumerations**

COUSE CONTENT

Collections

- List
- Map
- Set
- Aggregate Operations

- **Generics**

COLLECTIONS

- **What is a collection?**
  - A group of things that have been gathered [http://www.merriam-webster.com/dictionary/collection]

- **What is a collection in Java?**
  - Are containers of Objects which by polymorphism can hold any class that derives from Object
  - GENERICS make containers aware of the type of objects they store
    - from Java 1.5

COLLECTIONS. EXAMPLE. JAVA < 1.5

```
static public void main(String[] args) {
    ArrayList argsList = new ArrayList();
    argsList.add(args.length);
    for(String str : args) {
        argsList.add(str);
    }
    if(argsList.contains("Java") {
       System.out.println("Found Java word in collection");
    }
    String first = (String)argsList.get(0);
    System.out.println("First: " + first);
}
```

argsList Collection contains different types of objects:
- Integer
- String

Convert at the data type stored inside collection

COLLECTIONS. EXAMPLE. JAVA ≥ 1.5

```
static public void main(String[] args) {
    ArrayList<String> argsList = new ArrayList<String>();
    argsList.add(args.length);
    for(String str : args) {
        argsList.add(str);
    }
    if(argsList.contains("Java") {
       System.out.println("Found Java word in collection");
    }
    String first = argsList.get(0); // NO CASTING
    System.out.println("First: " + first);
}
```

ERROR way?
Static check of the collection objects type

No casting

GENERICS

- **Introduced in Java 1.5**
- **Allows class and methods definitions with parameters for types**
  - Classes or methods that have type parameters are called *parameterized class* or *generic definitions*, or, simply, *generics*
- **Can be defined by**
  - Java libraries
  - User

GENERICS CLASSES. EXAMPLES FROM JAVA LIBRARY

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
```

Interface for `List` class

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Interface for `Iterator` class, not used often explicitly

```
public interface Map<K,V> {
    V put(K key, V value);
}
```

Interface for `Map` class

GENERICS CLASSES. EXAMPLES. USER DEFINED

Generics Types T1, T2

DEFINITION

```
public class MyPair<T1, T2> {
  private T1 leftValue;
  private T2 rightValue;

  public MyPair (T1 t1, T2 t2){
     leftValue = t1;
     rightValue = t2;
  }
  public String toString(){
     return "(" + leftValue + ", "
               + rightValue + ")";
  }
  public T1 getLeftValue(){
    return leftValue;
  }
}
```

USAGE

```
MyPair<Integer, Integer> p1 =
          new MyPair<Integer, Integer>(10, 8);

System.out.println("p1: " + p1);

MyPair<String, Double> p2 =
          new MyPair<String, Double>("Coffe", 1.5);

System.out.println("p2: " + p2.getLeftValue());
```

GENERICS CLASSES

- **Syntax**
  - `class name<T1, T2, ..., Tn> { /* ... */ }`
- **Parameterized Types**
  - Can also substitute a type parameter (i.e., K or V) with a parameterized type
  - Example
    - `MyPair< String, ArrayList<Characters>> p3;`
- **Diamond operator**
  - <>
  - From Java ≥ 1.7
  - Can determine the arguments type from the context
  - Example
    - `MyPair< String, ArrayList<Characters>> p3 = new MyPair< String, ArrayList<Characters>>();`
    - Becomes: `MyPair< String, ArrayList<Characters>> p3 = new MyPair<>();`

GENERICS METHODS

- **Methods that introduce their own type parameters**
- **Can be static and non-static methods**
- **Example**

```
public class Util {
  public static <K, V> boolean compare(
                          MyPair<K, V> p1, MyPair<K, V> p2) {

    return p1.getRightValue().equals(p2.getRightValue()) &&
           p1.getLeftValue().equals(p2.getLeftValue());
  }
}
```

- Usage

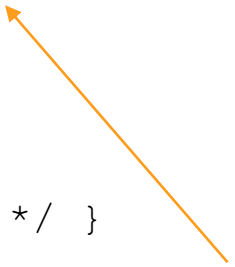
```
MyPair<Integer, Integer> p1 = new MyPair<Integer, Integer>(10,
8);
MyPair<Integer, Integer> p3 = new MyPair<Integer, Integer>(15,
8);
System.out.println("p1=p3? " + Util.compare(p1, p3));
```

GENERICS

- **Bounded Type Parameters**
  - Restrict the types that can be used as type arguments in a parameterized type
  - Example

```
public static <K extends Number, V extends Number>
  boolean compareJustNumbers(MyPair<K, V> p1,
                             MyPair<K, V> p2){…}
```

  - Also accepts multiple bounds
    - `<T extends B1 & B2 & B3>`
  - Example
    - `class D <T extends A & B & C> { /* ... */ }`



Usage

```
MyPair<Integer, Integer> p1 = new MyPair<Integer, Integer>(10, 8);
MyPair<String, Double> p2 = new MyPair<String, Double>("Coffee", 1.5);
MyPair<Integer, Integer> p3 = new MyPair<Integer, Integer>(15, 8);
MyPair<String, Double> p4 = new MyPair<String, Double>("Coffee", 1.5);
System.out.println("p1=p3? " + Util.compareJustNumbers(p1, p3));
System.out.println("p2=p4? " + Util.compareJustNumbers(p2, p4));
```

GENERICS

- **What happens when a generic type is instantiated?**
  - There is no real copy for each parameterized type (Unlike Templates in C++)
  - Compile time check (e.g. `List<Integer>` adds only `Integer` type objects)
  - Compiler adds run-time casting (e.g. pulling item from `List<Integer>` goes through run-time casting to `Integer`)
  - At run-time, the parameterized types (e.g. `<T>`) are erased – this technique is called Erasure
    - E.g. `List<String>` is converted to `List`
    - E.g. `String t = stringlist.iterator().next()` is converted to `String t = (String) stringlist.iterator().next()`

What would be the result of the following code?

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() ==  l2.getClass());
```

`True` no other class in created

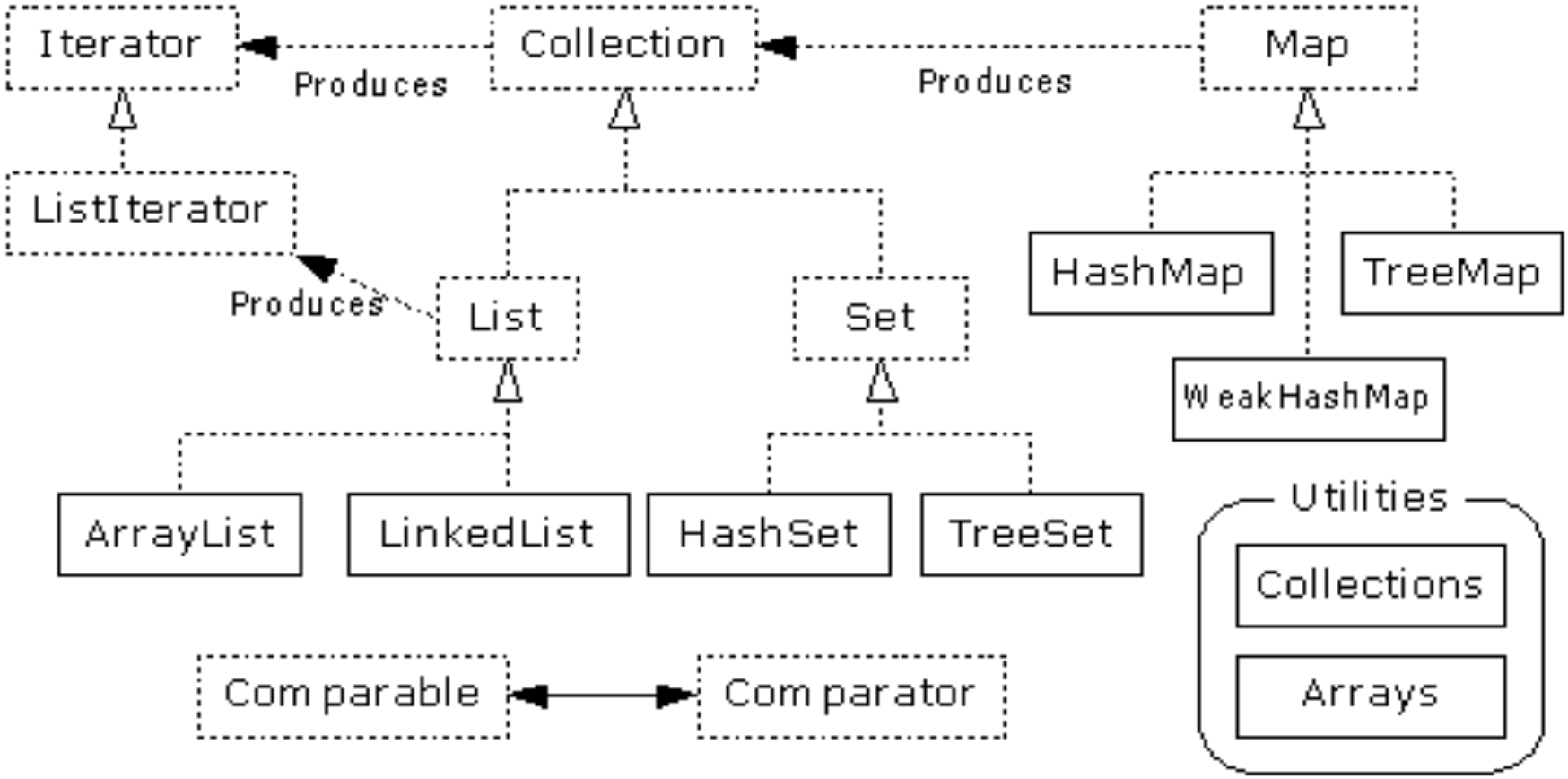
COLLECTIONS

- **A collection is an object that groups multiple elements into a single unit**

- **Java propose a collection framework**
  - Unified architecture for representing and manipulating collections

  - A collections framework contains three things
    - Interfaces
      - Each defines the operations and contracts for a particular type of collection (`List`, `Set`, `Queue`, etc)
      - Idea: when using a collection object, it's sufficient to know its interface
    - Implementations
      - Reusable classes that implement above interfaces (e.g. `LinkedList`, `HashSet`)
    - Algorithms
      - Useful polymorphic methods for manipulating and creating objects whose classes implement collection interfaces
      - Sorting, index searching, reversing, replacing etc.

COLLECTIONS



COLLECTIONS

- **Collection interface**
  - Defines fundamental methods
    - `int size();`
    - `boolean isEmpty();`
    - `boolean contains(Object element);`
    - `boolean add(Object element);` // Optional
    - `boolean remove(Object element);` // Optional
    - `Iterator iterator();`
  - These methods are enough to define the basic behavior of a collection
  - Provides an `Iterator` to step through the elements in the Collection

COLLECTION ITERATOR

- **An `Iterator` is an object that enables to traverse through a collection and to remove elements from the collection selectively, if desired**

- `iterator()` **method is used to obtain an iterator for a collection**

- `Iterator` **interface**

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

COLLECTION ITERATOR

- **Display a collection using an iterator**

```
List<String> list = new ArrayList<>();
list.add("course");list.add("four"); list.add("java");

for(Iterator<String> it = list.iterator(); it.hasNext();){
    System.out.println("List element: " + it.next());
}
```

- **Display a collection using for-each statement**

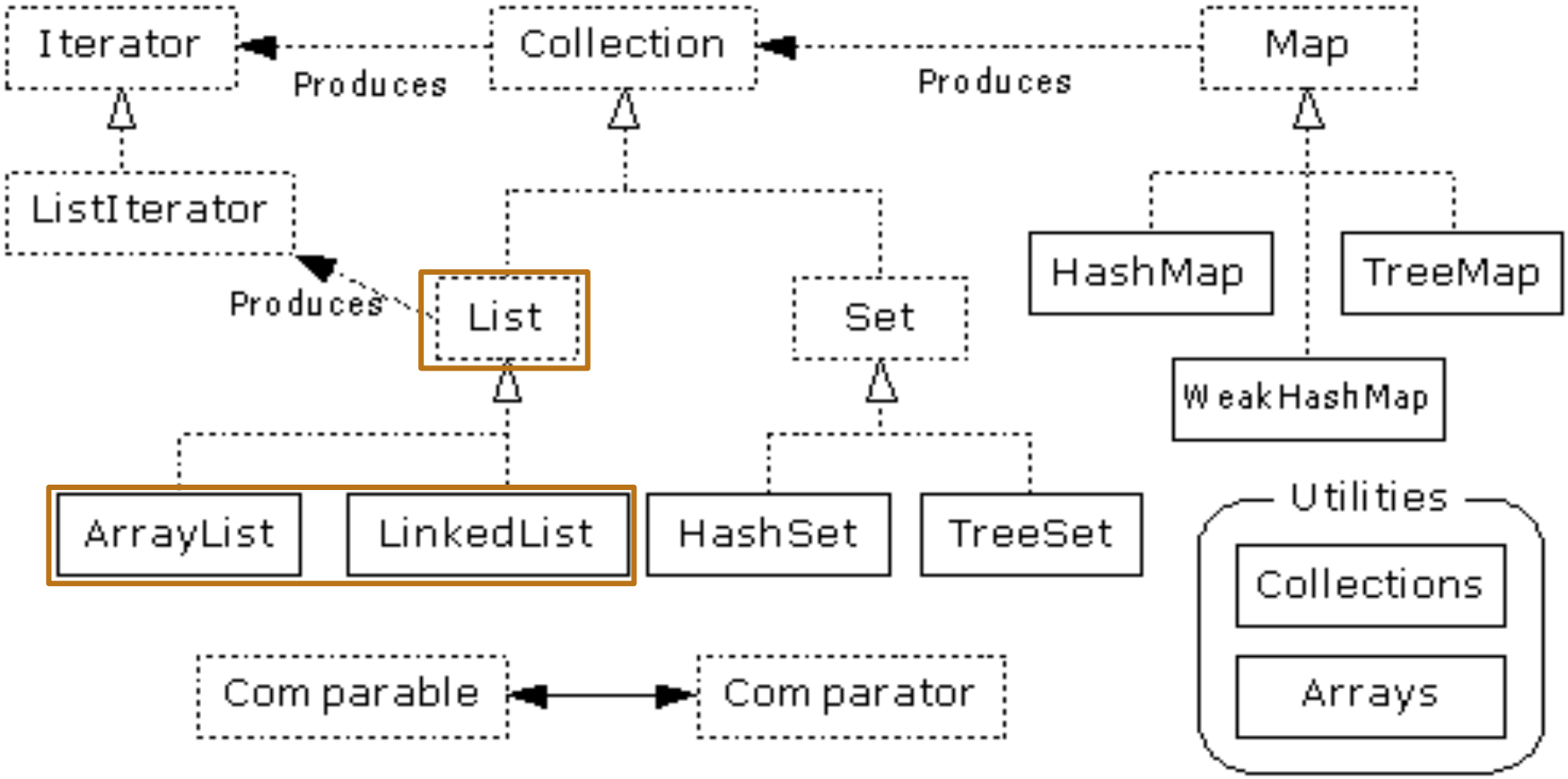
```
List<String> list = new ArrayList<>();
list.add("course");list.add("four"); list.add("java");

for (String element: list){
    System.out.println("List element: " + element);
}
```

For-each statement can be used by arrays also.
Disadvantage: losing the index position.
Example: `int a[]={1,2,3}; for (int el : a) System.out.println(el);`

There is no difference between the two implementation
For-each is translated similar with the first example

COLLECTIONS



COOLECTION. LIST INTFACE

- **Java provides 3 concrete classes which implement the list interface**
  - `Vector`
  - `ArrayList`
  - `LinkedList`

- `Vector` **try to optimize storage requirements by growing and shrinking as required**
  - Dynamic array
  - Methods are synchronized (used for Multi threading)

- `ArrayList` **is roughly equivalent to Vector except that its methods are not synchronized**

- `LinkedList` **implements a doubly linked list of elements**
  - Methods are not synchronized

COOLECTION. LIST INTFACE

- **A list is an ordered Collection (sometimes called a sequence).**
- **Lists may contain duplicate elements.**
- **In addition to the operations inherited from `Collection`, the `List` interface includes operations for the following:**
  - Positional access
    - Manipulates elements based on their numerical position in the list
    - Includes methods such as *`get()`*, *`set()`*, *`add()`*, *`addAll()`* and *`remove()`*.
  - Search
    - Searches for a specified object in the list and returns its numerical position.
    - Search methods include *`indexOf()`* and *`lastIndexOf()`*.
  - Iteration
    - Extends `Iterator` semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
  - Range-view
    - The `sublist()` method performs arbitrary range operations on the list.

COOLECTION. LIST INTFACE

- **Example**

```
List<String> a1 = new ArrayList<>();

a1.add("Course");
a1.add("Programming");
a1.add("III");

System.out.println(" ArrayList Elements");
System.out.print("\t" + a1);

List<String> a2 = new LinkedLis<>();

a2.addAll(a1);

System.out.print("Element on position 2 in list: " + a2.get(2));

a1.set(2, "Java");

a1.remove("Programming");

int i = a2.lastIndex("III");
```

Declare and initialize a list object

Add elements to a list

Display the list, `toString()` method is overwritten

Adds all elements of list `a1` into list `a2`

Get/set the element at index 2 from the list

Remove an element from the list

Search the las occurrence of an object in a list

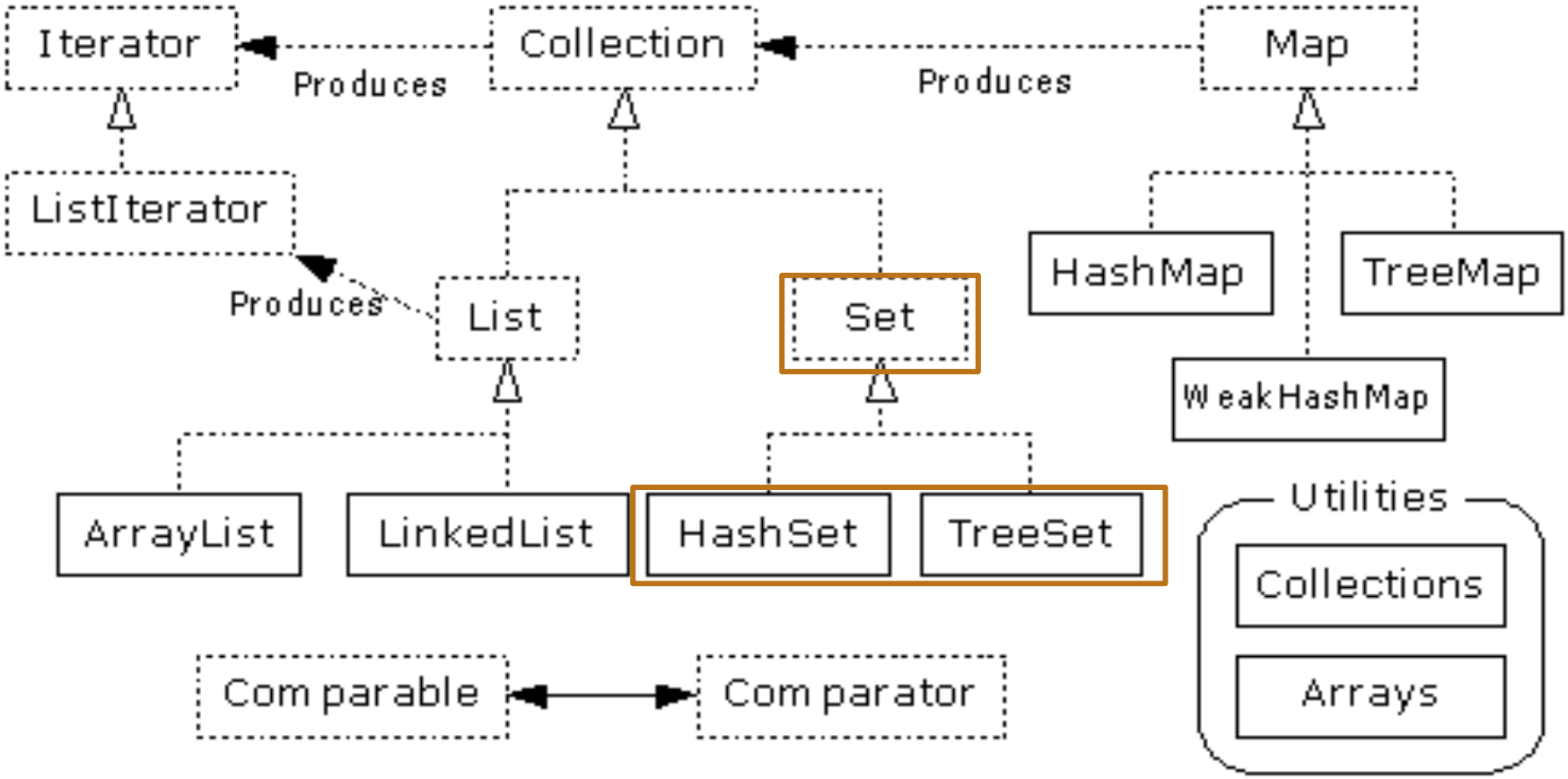
COOLECTION. LIST IMPLEMENTATIONS

- **Which type of List implementation to choose?**

  - `ArrayList`
    - dynamic array
    - low cost random access
    - high cost insert and delete
    - array that resizes if is need

  - `LinkedList`
    - linked list
    - sequential access
    - low cost insert and delete
    - high cost random access

COLLECTIONS



COLLECTIONS. SET INTERFACE

- **Java provides two concrete classes which implement the `Set` interface**
  - `HashSet`
  - `TreeSet`

- **The elements cannot be duplicated.**

- **The `Set` interface contains only methods inherited from `Collection` and adds the restriction that duplicate elements are prohibited.**

COLLECTIONS. SET INTERFACE

- **Example**

```
int count[] = {34, 22,10,60,30,22};

Set<Integer> set = new HashSet<Integer>();

for(int i = 0; i < 5; i++) set.add(count[i]);

System.out.println(set);

TreeSet<Integer> sortedSet = new TreeSet<Integer>(set);
System.out.println("The sorted list is:" );
System.out.println(sortedSet);

System.out.println("The First element of the set is: "+
(Integer)sortedSet.first());
System.out.println("The last element of the set is: "+
(Integer)sortedSet.last());
```

Declaration and initialization of a set object

Adds an element to a set

Obtain first/last element from a set

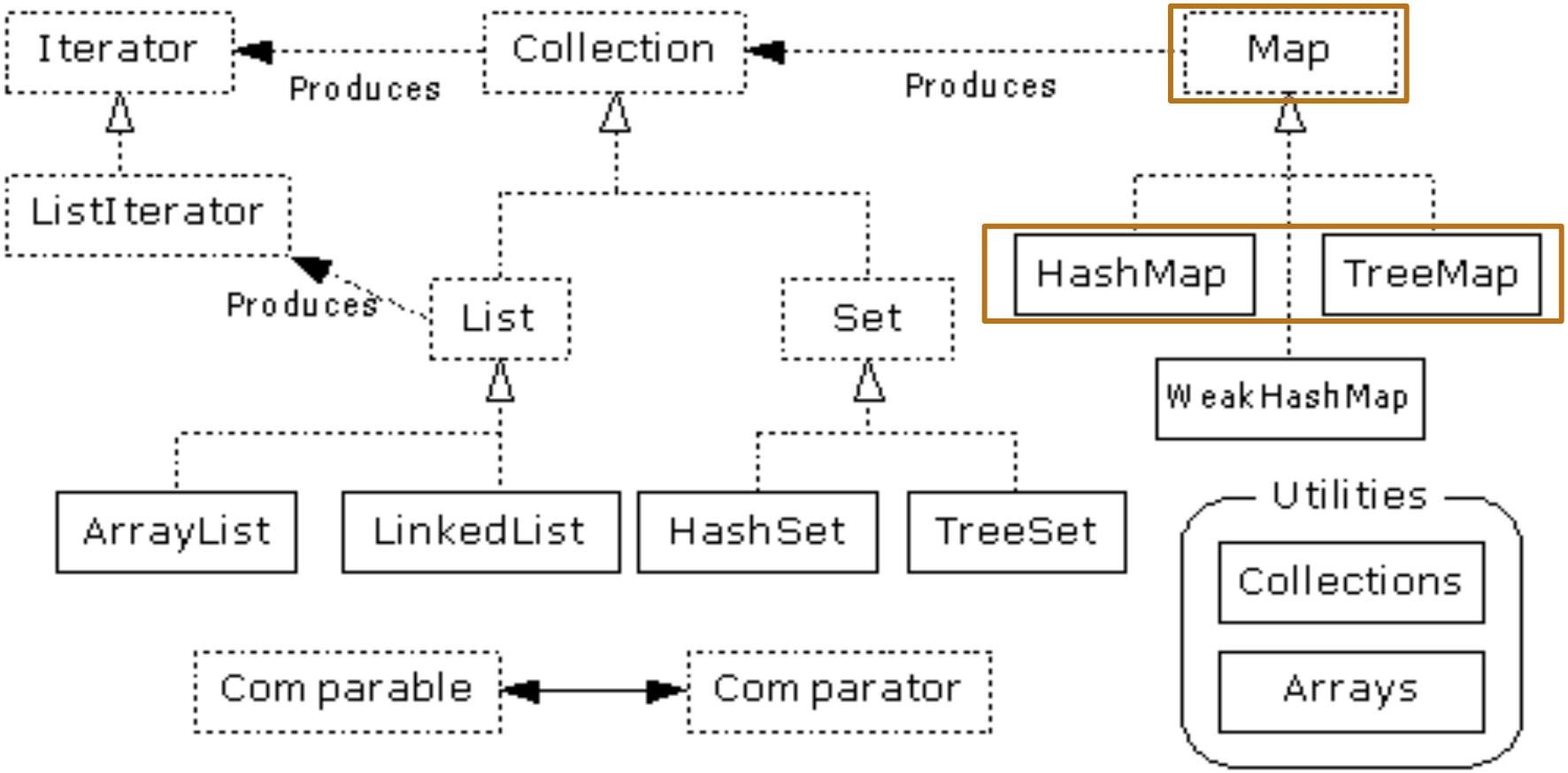
COLLECTION. HASHSET

- **Find and add elements very quickly**
  - uses hashing implementation in `HashMap`
- **Hashing uses an array of linked lists**
  - The `hashCode()` is used to index into the array
  - Then `equals()` is used to determine if element is in the (short) list of elements at that index
- **No order imposed on elements**
- **The `hashCode()` method and the `equals()` method must be compatible**
  - if two objects are equal, they must have the same `hashCode()` value

COLLECTION. TREESET

- **Elements can be inserted in any order**
- **The `TreeSet` stores them in order**
- **An iterator always presents them in order**
- **Default order is defined by natural order**
  - objects implement the Comparable interface
  - `TreeSet` uses *`compareTo(Object o)`* to sort
- **Can use a different `Comparator`**
  - provide `Comparator` to the `TreeSet` constructor

COLLECTIONS



COLLECTION. MAP INTERFACE

- **Stores <key, value> pairs**
- **Maps from the key to the value**
- **Keys are unique**
  - a single key only appears once in the Map
  - a key can map to only one value
- **Values do not have to be unique**

COLLECTION. MAP INTERFACE

- **Operations**
  - `Object put(Object key, Object value)`
  - `Object get(Object key)`
  - `Object remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`

COLLECTION. MAP INTERFACE

- **Iterating over the keys and values in a Map**
  - `Set keySet()`
    - returns the `Set` of keys contained in the `Map`
  - `Collection values()`
    - returns the `Collection` of values contained in the map.
    - this `Collection` is not a `Set`, as multiple keys can map to the same value.
  - `Set entrySet()`
    - returns the *`Set`* of key-value pairs contained in the *map*.
    - *Map* interface provides a small nested interface called `Map.Entry` that is the type of the elements in this *Set*.

COLLECTION. MAP INTERFACE IMPLEMENTATIONS

- **`HashMap`**
  - The keys are a set - unique, unordered
  - Fast

- **`TreeMap`**
  - The keys are a set - unique, ordered
  - Same options for ordering as a `TreeSet`
  - Natural order (`Comparable, compareTo(Object)`)
  - Special order (`Comparator, compare(Object, Object)`)

MAP. EXAMPLE

- **Exercise**
  - Create a map thet contains the number of appearences of a letter into a word.
    - ex: “maria”
      - m - 1 times
      - a - 2 times
      - r - 1 times
      - i - 1 times